ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs

Min-Young Nam, Rodolfo Pellizzoni, Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
Champaign, Illinois, U.S.A.
Email: {mnam, rpelliz2}@uiuc.edu, lrs@cs.uiuc.edu

Richard M. Bradford
Rockwell Collins Inc.
Cedar Rapids, Iowa, U.S.A.
Email: rmbradfo@rockwellcollins.com

Abstract

In hard real-time systems such as avionics, computer board level designs are typically customized to meet specific reliability and real-time requirements. This paper focuses on computer-aided application-specific design of I/O architecture using PCI as an example. We have built a tool (ASIIST) that will enable engineers to explore design spaces at the I/O bus architecture level, performing analysis that incorporates bus protocols, to provide guarantees of real-time properties.

1. Introduction

In hard real-time systems such as avionics, computer board level designs are typically customized to meet specific reliability and real-time I/O requirements. In the design and development process, system level performance analysis should precede implementation in order to save cost and to optimize the I/O architecture.

Traditionally, system-level schedulability analysis has been done manually which is error prone and tedious. In the past few years, schedulability analysis tools that use systematic approaches to generate equations have been developed. The analysis algorithm they use safely bounds the worst case usage of resources, e.g. processing or communication. We have been collaborating with researchers and production engineers from Rockwell Collins in the development of an analysis framework and prototype computer-aided I/O analysis tool called ASIIST (Application Specific I/O Integration Support Tool). ASIIST defines hardware modeling blocks that have one-to-one correspondence with actual hardware components for the ease of understanding and use. Still, the underlying algorithm is modular and capable of analyzing alternative system architectures represented by the composition of the modeling blocks.

A typical large passenger airplane has hundreds of processors and thousands of software tasks. In a typical avionics company, the software and hardware engineers engaged in design and development outnumber the real-time analysts several hundreds to one. The current approach of assigning a few real-time analysts to interview engineers and manually build schedulability models cannot support the need to rapidly evaluate different real-time related design alternatives. To compound the difficulty, the given logical data flows may also be modified during the design process and the change of work load flows requires corresponding changes in schedulability and reconfigurations in I/O architecture. The system design architects are generally not specialized in building the tool specific schedulability models by themselves from the modeling blocks a tool provides, especially for large complex systems. As a result, the system level schedulability analysis often lags way behind of the actual designs to be done by the few real-time analysts. As one senior engineer has remarked,

“Architecture performance is a type of problem that is commonly injected during the design of an architecture but rarely identified at this stage - usually the problem surfaces during integration, where it is difficult and expensive to remedy.” [15]

Fortunately, the avionics industry is moving towards the adoption of architecture description language, SAE AADL [3] to support model based system engineering process. An AADL model is more of an architecture centric model than an equation centric model. This is possible because AADL lets the user model the hardware platform separately from the software logical model. Thus, the hardware modeling blocks have a one-to-one correspondence with the actual hardware components, which makes the architect’s life easy. Although AADL provides separation of hardware and software specification, there still remains...
the job of defining modular reusable abstractions on top of AADL components for selected bus components to support analysis. The definition should be made with the following two goals in mind:

- The model abstractions should correspond to actual hardware building blocks so that they are easy to use and reuse by system designers.
- The abstractions should be limited to a set of components that have well-defined real-time behaviors and can be scaled up for complex I/O-intensive real-time applications.

From the point of view of the architect, alternative designs should be easily expressed and analyzed. Thus, careful consideration needs to be taken when defining the level of abstraction used for the modeling blocks of a tool. It should be expressive enough to model the subset of the concerned architectural specifications while only utilizing judiciously selected components that are suitable for hard real-time application.

Figure 1 shows an example of a system which is a network in the form of a tree of bridges and bus segments shared by processors, storage elements, and I/O devices, using a PCI bus for communication. For each possible given hardware architecture, there exist different types of bus transactions and different ways to route data flows. Any change in the tree architecture, the selection of bus transaction types, or routes will create a different set of complex schedulability equations that the user must analyze. With the frequent architectural modifications that will occur to check new alternatives, a modular and architectural level of abstraction such as the modeling blocks shown in Figure 1 needs to be defined for the ease of use and to reduce human error for even the well trained user.

As a comparison, tools that are built to analyze heterogeneous systems ([22], [10]) have the benefit of being capable of supporting a larger set of possible architectures. However, the price to pay is having to analyze existing bus protocols and complex scheduling policies in order to translate them into the general modeling blocks required by the tool. For instance, a PCI delayed read bus transaction actually needs request messages to be sent before the actual data is transferred. The standard PCI protocol imposes a mixed scheduling policy of round robin and fixed priority scheduling. All these additional messages and implicit scheduling policy need to be considered for safety. This translation requires a fair amount of learning curve and is still prone if done manually because it is very tedious. These complications refrain a user from making frequent architectural changes which may result in a considerable amount of rework in modeling the changes.

By using an extended AADL model as the input for ASIIST, annotated AADL designs of software tasks, communication flows and hardware designs will be used to automatically generate schedulability equations. These analysis results are then used to support application specific hardware architectural design decisions. Figure 1 and Figure 2 actually show the level of abstraction for our tool to work which is considered to be user friendly in the architect's perspective.

This paper will focus on presenting a framework including our tool named ASIIST. As a first step to show its usefulness, we focus on a specific communication architecture, PCI, simply because PCI is very common and it is typical of peripheral interconnections. The analysis we support are the use of different bus architectures, example is shown in Figure 1, as well as changing I/O configuration as described in Figure 2 with PCI based components and bus transactions.
The main contribution of this paper is that it introduces a tool (ASIIST) that can analyze arbitrary bus architectures and apply commonly used bus protocols (PCI), using pre-defined AADL components which are convenient to the general system architect, while being extensible and scalable. We also define a computer-aided I/O design framework for building application specific bus architecture designs, where the framework can be extended with new user-defined analysis algorithms as well as bus architecture design algorithms. The prototype tool can be downloaded from https://agora.cs.uiuc.edu/display/realTimeSystems/ASIIST.

The rest of the paper is organized as follows. In Section 2, we give an overview of our I/O design framework. Section 3 gives an overview of AADL, how we define modeling blocks, how we specify data flows, and the extensions we have made to AADL. Section 4 gives a short introduction to the algorithm we are using to analyze PCI buses. In Section 5 we explain the architecture of ASIIST and the benefits of using it. Section 6 gives a simple heuristic algorithm as an example for building application specific bus architectures and Section 7 demonstrates the use of ASIIST with the heuristic algorithm for an example model. Finally, Section 8 explains other related work and Section 9 concludes this paper.

2. Computer-aided I/O Design Framework Overview

In this section, we will itemize the components of our framework to show the big picture. The framework consists of the following five abstractions.

- **Modeling of Bus Communication using AADL:** We define abstractions on top of AADL to better support the modeling of bus communication for ease of use. To support a certain protocol, such as PCI, protocol specific modeling blocks are defined. By defining these blocks, we also restrict users from applying the analysis to invalid models which are not suitable for hard real-time systems. Other bus protocols can be supported in a similar manner.

- **Bus Analysis Algorithm:** For ASIIST to be able to analyze an AADL model, we need a bus analysis algorithm which can incorporate an already made bus protocol. In Section 4 we briefly introduce a candidate PCI analysis algorithm that ASIIST is supporting which can compute worst case delay of data transfer and buffer backlog of bridges. However, ASIIST can be extended to use new algorithms for bus analysis as long as it can be applied to the modeling blocks.

- **ASIIST (Application Specific I/O Integration Support Tool):** ASIIST is a tool that reads input from a system model specified in AADL and can perform delay analysis of data flows that exist for a customized hardware platform. Having a tool which will read large system designs and build complex scheduability equations to solve is critical to the development of hard real-time systems.

- **A Heuristic Algorithm for Bus Architecture Design:** Even with the information of data flows that exist for a set of applications, it is an open research problem to derive an algorithm that generates the best bus architecture. Thus, we introduce a simple heuristic algorithm for building the bus architecture that can better satisfy the real-time requirements of the data flows. ASIIST gives a greater flexibility in coming up with a better heuristic algorithm because we are able to analyze the real-time performance of the architecture generated by the heuristic algorithm in a relatively quick manner. An example of a simple heuristic algorithm will be introduced in Section 6.

3. Modeling of Bus Communication using AADL

We will give a brief overview of AADL in this section and show how it is used or extended to specify bus communication, using PCI as an example. This includes the modeling blocks for the hardware architecture, specification of logical data flows, PCI protocol specific hardware flows, and I/O configuration options.

3.1. Overview of AADL

AADL is a language based on 15 years of research, including the MetaH language developed by Honeywell Labs and several DARPA programs [5], and it has been applied to many industry examples. Currently its development is led by Peter Feiler at the Software Engineering Institute (SEI). AADL provides a means of specifying the hardware and the software architecture of embedded systems. With AADL, we are able to perform various kinds of analyses, such as safety analysis, fault tolerance, schedulability, system latency, etc. It provides textual and graphical interfaces to allow users to build the architecture of a system composed of “components” and to specify the interaction among the components. The architecture also supports the development of tools that perform dynamic or static analysis based on the system specification.

3.1.1. AADL Components. AADL Components are divided into three categories: software, execution platform, and system. Software components include process, thread,
thread group, subprogram and data components. To represent the infrastructure, we have the processor, memory, bus and device as execution platform components. The system component is a special composite component which is used to divide components into groups or encapsulate components to distinguish them from others as a separate system object. It could also be used to substitute any new abstract entity that is not pre-defined above. Software components are bound to execution platform components to represent where it is executed, for threads, or where it resides, for data components.

Property specification plays a major role in AADL since properties add extra information that cannot be expressed by structural descriptions. The core AADL has standard pre-declared properties that support real-time scheduling as well as other areas of research. All bindings are done by property associations in AADL. AADL also provides the syntax to add new user-defined properties.

AADL is also extensible so that we can add sub-languages as annexes to describe more complicated semantics that can be processed as part of the specification of an AADL model.

3.1.2. Modal Specification. A system can operate in different modes. Each mode can be represented in the form of an AADL modal specification. In the manner, property assignment can include mode specification to restrict the validity of the property value to certain operational modes. Thus, modes can be used to represent alternative hardware flow paths for a single logical data flow connection. For multiple mode specifications among different connections, the OSATE API provides convenient interfaces to check for every combinatory instance of modes. More information about AADL can be found in [8] or at http://www.aadl.info.

3.2. Modeling Block Definition for PCI Bus Architecture

The general method of defining modeling blocks in AADL is using the standard components that AADL provides. However, AADL does not include every specific kind of component we may encounter during development. For example, AADL does not have bridges, switches, registers, or batteries. However, we would need to identify these types of components if they are to be used for analysis. These non-standard components are represented by using available component types in AADL that have a somewhat similar syntax. For example bridges can be represented by device components because they would be connected to buses and should be considered as a hardware component. If nothing matches the syntax, there is always the system component, which can be used in almost any possible syntax.

3.2.1. Basics of PCI Bus Standard. Before explaining the extensions we made for PCI, it would be beneficial to go over the basics of the PCI protocol. The Peripheral Component Interconnect (PCI) is the current standard family of communication architectures for motherboard-parallel interconnection in the personal computer market; it is also widely popular in the embedded domain [2]. The standard can be divided into two parts: a logical specification, which details how the CPU configures and accesses peripherals through the system controller, and a physical specification, which details how peripherals are connected to and communicate with the motherboard. In this section we focus on the PCI/PCI-X physical specification, which uses a shared bus architecture with support for multiple bus segments connected by bridges. A typical PCI-based platform is shown in Figure 1. The CPU and main memory are connected through the Front Side Bus (FSB), which has a CPU-manufacturer dependent implementation. A host bridge is also connected to the FSB and offers access to the rest of the system using the PCI standard. Each further PCI-to-PCI bridge connects two PCI bus segments together. The architecture resembles a tree, where the host bridge represents the root, bridges are intermediate nodes, and peripherals represent leaves. Data transfers are carried out on each bus segment as non-preemptive bus transactions; the entity that starts a transaction (either a peripheral or the CPU) is known as the initiator, while the entity that receives the transaction (either another peripheral or a memory) is known as the target. Each bus segment has a separate arbiter which determines the order in which initiators are allowed to transmit. If the initiator and target of a transaction reside on different bus segments, then the transaction data is stored and forwarded by intermediate bridges; note that due to the tree-shaped structure of PCI, there is a single path between any two bus segments. The PCI standard offers support for several types of both read and write transactions. Please refer to [2] for details on the PCI bus standard.

3.2.2. Modeling Blocks for PCI. PCI bus is not designed for real-time systems. Since our approach is to develop modeling blocks corresponding to architecture building blocks for hard real-time applications, the selection of building blocks, configuration rules and transaction types are important. After receiving hardware architecture models from industry companies, it became natural to define modeling blocks that represent the PCI bridge and PCI bus, as shown in Figure 1. These are pre-defined in AADL so that users can use them or extend them. Using these modeling blocks lets us express any tree shape of PCI bus architecture and easily understand the architecture. AADL allows components to be extended. We use extension for checking the applicability of an analysis. An analysis could distinguish between a
general and an extended component. In this case, the tool will pass the extended component to the analysis. If the analysis does not care, then the tool passes it as a general component to the analysis. In our examples, host bridge and FSB will be modeled with general bridge and bus components because the analysis described in Section 4 does not consider any extension of these kinds (host bridge and FSB). Analysis supports general components by assuming worst cases scenarios (see Section 4).

3.3. Logical Data Flows in AADL

Logical data flows are the exchange of data which is required by applications executing in a system. AADL plays an important role of specifying logical data flows. Figure 2 (a) is a graphical example of a logical flow specified using AADL. It is independent of the hardware through which the data is physically transmitted and any protocols that are applied to the hardware components. Properties of a logical data flow include but are not limited to data size, period, deadline, source, and destination.

The standard AADL already provides a reasonable syntax for expressing the interactions between software components by using event, data, or event data ports and connections. AADL flows can be specified to express a sequence of connections to trace data or control. Connections can also be bound to buses to analyze the effect of the I/O bus architecture.

3.4. Extension for H/W Flow Descriptions

A logical data flow, in reality, will go through multiple hardware components to get to a destination. If the source and destination applications run on different CPUs that are located is separate systems, it would even go through a network. Depending on the type of the hardware used, the data is regulated to experience the already defined protocol for the COTS components it goes through. A designer can also make I/O configurations, as shown in Figure 2, to improve system performance. The real system performance is dependent on these hardware flows that are derived from the logical data flows and I/O configurations.

The standard AADL provides a property to specify the hardware flow path for a port connection. The property Actual_Connection_Binding is used for this purpose and can have a list of execution platform references as its value to specify the execution platform resources that are to be used for the logical data flow. For the example shown in Figure 2 (b), we could assign a list value of “Peripheral, PCI Bus, Bridge, FSB, CPU” to represent the hardware flow path of the data.

property set hardwareflow is
Write_Actual_Connection_Binding: inherit list of reference
(bus, processor, device, memory, system) applies to
(port connections, thread, thread group, process, system);
PCI_Write_Type: inherit enumeration (Posted_Write, Delayed_Write)
applies to (port connections, thread, thread group, process, system);
PCI_Write_Period: inherit Time applies to
(port connections, thread, thread group, process, system);
end hardwareflow;

Read_Actual_Connection_Binding: inherit list of reference (bus, ...);
PCI_Read_Type: inherit enumeration (Delayed_Read) applies to ... ;
PCI_Read_Period: inherit Time applies to (port connections, ...);
end hardwareflow;

Figure 3. Hardware Flow Property Definition

However, just using the Actual_Connection_Binding property is not expressive enough to represent the various multiple hardware flows that will be used for a single data port connection. For Figure 2 (c), we may try considering a list value of “Peripheral, PCI Bus, Bridge, FSB, System Memory, FSB, CPU” which somewhat expresses the fact that the data goes through the system memory. However, this is invalid for the standard AADL property Actual_Connection_Binding because it includes a memory reference which is not allowed. The description also does not specify in what period the peripheral writes to the system memory or what kind of PCI transaction it uses. In the following, we will extend AADL to address these issues in describing H/W flows for PCI protocol.

3.4.1. Extensions to AADL for PCI Protocol. We have defined a set of properties to extend AADL to be capable of describing the hardware flows needed for our new analysis. Adding new properties to AADL is a common practice for new tool developers because additional information needed may be missing in the standard.

Figure 3 describes the properties that we have newly defined. We used the fact that the generated set of hardware flows can be categorized into write transactions and read transactions. Write_Actual_Connection_Binding and Read_Actual_Connection_Binding are used the same way as Actual_Connection_Binding except for the fact that they hold the list of execution platform components for each write or read hardware flow. Depending on the protocol that the PCI bus provides, PCI_Write_Type and PCI_Read_Type can have the values Posted_Write, or Delayed_Write, and Delayed_Read respectively. These are the transactions that can be supported for hard real-time analysis. Previous versions of PCI transactions, before 3.0, which utilize blocking should not be used for real-time systems because transaction delay can be extremely large in the worst
case. Delayed_Write is used when an initiator would like to get acknowledgements that the write transaction has finished. Useful extensions can eventually evolve into annexes and can also be standardized. Although AADL has a standard property named Period which is used to represent the period of a task, when there are multiple hardware flows, each hardware flow could have a different period. Thus, we have included the properties PCI_Write_Period and PCI_Read_Period to specify the period of each hardware flow.

3.5. Using Modal Specification for I/O Configuration

As we mentioned in Section 3.1.2, design decisions that can be expressed as an AADL property value can be preserved by using modal specification of the property assignment. Figure 4 shows how we can express the three different I/O configuration options introduced in Figure 2. The three types of configurations are given mode descriptions (option1, option2, and option3) to represent the options. The period value for each hardware flow can be assigned if it is different from the period of the logical data connection. The OSATE API will automatically generate different model instances for analysis based on the collection of available sets of modes for different data connections.

![Figure 4. Modal Property Assignment Example](image)

4. PCI Analysis Support

In this section, we will give a brief summary of the analysis algorithm that ASHIST currently supports for PCI bus analysis.

The simplest type of PCI bus transaction is posted write. A posted write completes at the initiator before it completes at the target: in other words, after data has been moved from the initiator to the first intermediate bridge, the initiator can disconnect from the bus and considers the transaction successfully completed. In this way, data is posted from bridge to bridge until the target is reached, similarly to what happens in a packet-switched network. As a matter of fact, we base our analysis on network calculus theory [6], which is able to compute deterministic delay bounds for network traffic and has also been applied to model different types of real-time systems [21]. We treat all system communication as a set of $N$ flows $\{f_1,\ldots,f_i,\ldots,f_N\}$, each between a specified source (an initiator) and destination (a target).

Network calculus considers a system comprised of multiple network elements, or routers, connected by point-to-point lines, but the PCI architecture employs shared buses. We can still rely on the network calculus model using the following transformation: we treat each bus segment, together with all bridges that transmit data on it, as a unique network element. Network elements are then connected if the corresponding bus segments are interconnected by a bridge.

A clarifying picture is shown in Figure 5, for a flow $f_i$ that goes through 4 bus segments. The flow is divided into a set of $N$ subflows, $f_i^1,\ldots,f_i^4$ representing the traffic generated by the initiator and $f_i^1,\ldots,f_i^4$ being the traffic on each subsequent bus segment. An arrival curve $\alpha_i^j(t)$ is associated with each subflow $f_i^j$. $\alpha_i^j(t)$ is the worst-case amount of traffic of $f_i^j$ that can be transmitted in any time interval of length $t$. Let $B_k$ be the network element that outputs $f_i^j$; network calculus provides a way to relate $\alpha_i^j(t)$ to the arrival curves of all subflows that are inputs to $B_k$, and to compute the maximum delay suffered by any byte of $f_i$ that traverses $B_k$.

The input-output relation can be simplified by adopting
a linearized representation for arrival curves. We consider an upper bound to the arrival curve of the form \( \delta_i^j + \rho_i^j \), where \( \delta_i^j \) represents the burstiness and \( \rho_i^j \) the arrival rate for \( \alpha_i^j(t) \). An example of linearized arrival curve \( \alpha_i^j(t) \) for a strictly periodic initiator that generates \( e_i \) bytes of traffic every \( p_i \) time units is shown in Figure 6. Based on linearized arrival curves we can then prove the following (see [16] for details): if the system is stable, the arrival rates \( \rho_i^j \) for all subflows of \( f_i \) are equal, therefore we use \( \rho_i \) as the arrival rate for \( f_i \). The burstiness of each subflow can be computed according to the following corollary.

**Corollary 1:** Let \( B_k \) be the element that outputs subflow \( f_i^j \), and \( \text{inter}_l \) be the set of subflows that are inputs to \( B_k \) (with the exception of \( f_j^{-1} \)) and \( C \) be the speed of the bus. Then \( \alpha_i^j(t) \) is bounded iff \( \sum_{j \in \text{inter}_l} \rho_l + \rho_i \leq C \), in which case:

\[
\delta_i^j = \delta_i^{j-1} + \rho_i \sum_{j \in \text{inter}_l} \delta_l^j \frac{\delta_l^j}{C - \sum_{j \in \text{inter}_l} \rho_l}.
\]

(1)

The inequality \( \sum_{j \in \text{inter}_l} \rho_l + \rho_i \leq C \) expresses the necessary condition that the sum of the rates of all incoming flows to a bus segment must be smaller than \( C \), i.e. the bus segment utilization must not exceed one. Using Corollary 1, we can write a system of equations, one for each subflow. It can be shown that the resulting system is linear in the burstiness variables \( \delta_i^j \), therefore it can be easily solved. Given subflow burstiness, worst case delay for each flow can then be computed.

The burstiness bound of Corollary 1 holds for any bus segment, since it is based on the pessimistic assumption that \( f_i \) has lower priority than all other flows. However, if we know more about the implementation of bus segment arbiters and bridges we can do better. In particular, if a bridge employs FIFO buffering for flows moving in the same direction, then the bound can be significantly improved. Similarly, if the bus arbiter employs round-robin arbitration, we can derive a second possibly less pessimistic bound. Due to space constraints, we do not elaborate on such extensions here; instead, we refer the interested reader to our companion technical report [16] which includes full details of the analysis. Finally, note that read transactions add further complexity to the analysis: a read request must reach the target before the data is sent back to the initiator. Once again, in [16] we show how delayed read transactions can be incorporated in the analysis adding additional forward and backward flows in the model.

5. Application Specific I/O Integration Support Tool

ASIIST (Application Specific I/O Integration Support Tool) is a plug-in tool that is added to the OSATE working environment to support application specific design of PCI based I/O architectures.

5.1. ASIIST Architecture

The architecture of ASIIST is shown in Figure 7. The thick-lined squares represent the inputs that need to be provided to ASIIST. The OSATE API provides the AADL instance model of the system. The user can configure three additional inputs to have ASIIST perform a customized bus analysis. All of the following configurations are stored as an XML [1] file that will be filled out through a graphical user interface. This helps assure that no invalid input is given.

- **HW Component Identification Rules:** Previously in Section 3.2.2 we have explained that extension is used for modeling blocks and that it is also used to identify the components that an analysis would support. To identify any component that is needed by the analysis, identification rules need to be expressed explicitly to assure the user that the model is read correctly into the tool. An example rule would be to check component type identifiers.

- **Bus Protocol Validity Checking Rules:** For any analysis, there are assumptions about the structure of the model in hand. For a PCI bus protocol, we would have to assume that each PCI bridge can only have two buses and that the structure is a tree. This and many other conditions need to be satisfied for the analysis to hold. It is much less error prone to have these structural conditions checked prior to running the actual analysis. Also, a user interface guides the user in defining the rules and will confine the structural aspects of the model to instances that are valid for analysis.

- **Bus Analysis Configuration:** Bus analysis configuration provides the optional settings of the actual bus analysis. When developing an analysis mechanism, we often confront situations where more than one method of analysis could be valid. One method might be better for certain situations, while another may be better for
others; one method may give a better result but is applicable for only a subset of the possible models. Bus analysis configuration represents the various optional methods available for general cases and shows the ones covered by the current tool.

With the above inputs for the tool, we will briefly explain the rest of the architecture shown in Figure 7. Using the AADL instance model and the component identification rules, the AADL component handler will pass on a component to the HW component builder. The bus protocol builder will have all the data to verify that the instance model is valid when the mode is interpreted with the HW component builder. With the combination of the two builders, we will have a model, specific to bus analysis, that can be analyzed by the bus delay analyzer. For any complicated mathematical computation, such as matrix inversion, we use the Mathematica Kernel [4] as a computational engine. Other reasons such as generating parametric equations, which is not covered in this paper, are involved in the decision of using Mathematica. In order to use the Mathematica Kernel from a Java environment, we use a toolkit named J/Link to communicate mathematical expressions to the kernel and get the answers back. Using Mathematica enables us to solve linear equations easily and to symbolically get solutions for unknown parameters. The user can modify the database after it is initially loaded from the AADL instance model to try new parameter values, such as period values, to manage the delay factors of all the hardware flows. To assist the decisions on the periods, the user can change the value of a period into a variable to see how it is related to delay values as an equation. Finally, when the user has a satisfiable selection, we reload the changes back to the AADL instance model for update.

We have designed the architecture of ASIIST to allow users to specify preferences within a type of bus protocol and bus analysis algorithm that is to be used. Since the actual bridge implementation is flexible for manufacturers, this architecture also allows the user to configure the model to match the actual implementation of the hardware as closely as possible and check whether it is a supported model for the analysis. Checking for the correct models and using the correct analysis method is very important and should be automated in case users are not familiar with the details of buses.

### 5.2. Benefits of using ASIIST

The following summarizes the benefits of using ASIIST. More information can be found in an extended version of this paper [14].

- **Evaluation of Larger Design Spaces:** Using ASIIST, a designer can find the real-time I/O performance as soon as (s)he enters a design alternative into annotated AADL, investigating a much large number of design alternatives and find the near optimal one before committing to implementation.

- **Extensibility and Scalability:** Extensibility is achieved by using isolated layers of data structures (logical data flow, H/W data flow, analysis specific data flow) involved in the process of analyzing a system model for I/O analysis. Bus protocols and analysis algorithms act as the interface between the layers. Scalability is achieved by managing the change in data flows to identify situations when new analysis can be localized and does not have to be rerun for the whole system.

- **Low Overhead of Usage and Reduction of Human error:** ASIIST is benefited by the current movement in avionics of adopting AADL to model their systems. The example models that we introduced in this paper are modified and simplified models that we have received from Rockwell Collins. Thus, there is less overhead of building a separate model in AADL for this particular analysis. Automated equation generation reduces human error caused by misinterpretation or simple human mistakes.

### 6. A First Fit Decreasing (FFD) Heuristic Algorithm for Bus Architecture Design

In this section, we will give a summary of a heuristic algorithm that can utilize ASIIST to come up with an application specific design for a PCI based I/O architecture. Building the best bus architecture, based on the applications that use the bus, is an open research problem. Thus, we give a possible algorithm to show how a heuristic algorithm can be made. The evaluation of this heuristic algorithm is out of the scope of this paper. We do note, however, that the
availability of ASIIST greatly facilitates the development of such heuristic algorithms for solving this problem. Of course, ASIIST can be used as part of any new heuristic algorithm that a user would like to create. Later in the next section, we will demonstrate that we can improve a concrete design case by using the heuristic.

We first define a measure of allocation priority. Allocation priority represents the difficulty of scheduling a data flow. It is used as the metric for ordering the data flows to schedule first. Flows that have larger data and transfer frequently would have a higher allocation priority. Also, if the speed of the bus that it goes through is lower, it would be more difficult to schedule that flow. Another related factor is the deadline requirement of the flow. For data flows, relative deadlines can be larger than the period of the flow. Shorter deadlines will increase the allocation priority. Considering all the above, we can express allocation priority as the following for all logical data flows,

$$ w \cdot \frac{L}{d \cdot C} $$  

Step 1: For a set of flows that share a bus segment, find the two flows that have the highest allocation priority and can have parallelism between each other, in other words, they do not have a common initiator or target. Divide the two flows to separate sets. We do not consider the flows that go through the upper bridge.

Step 2: For the rest of the flows, following the order of the allocation priority, assign flows to one of the sets if possible. In doing so, if one of the initiator or target is already in one set, it should be added to the same set. A flow is not possible to be assigned if the initiator and target of the flow are already in different sets.

Step 3: Compare the two sets to select who gets direct access to the upper bridge. The selection criteria can be flexible. It could be the sum of all the allocation priority of the flows that goes through the upper bridge. Another criteria would be only to compare the maximum allocation priority of the flows that goes through the upper bridge for each set of flows. Depending on the selection, we can extend the tree as shown in Figure 8 (d) for increased parallelism. In cases when the two sets are comparable within a bound, we can use a different type of extension as in Figure 8 (e).

Step 4: Use ASIIST to check if all the deadlines are met for all flows. If not, do step 1 for the set of flows which has a flow that missed its deadline.

If the above sequence does not produce a feasible solution, increase $w$ of the flow that misses the deadline by a factor of 2 and retry the process.

7. ASIIST Demonstration

This section gives a demonstration of using the FFD algorithm and rebuilding the bus architecture for the data flows introduced in the example shown in Figure 8. ASIIST is used to compare the architectures shown in Figure 1 of the introduction.

7.1. Example Model

For the example model, we will use the previous hardware platform that was presented in Figure 1. Figure 1 (a) is considered to be an initial bus architecture and Figure 1 (b) is the resulting bus architecture after using the FFD heuristic algorithm. Table 1 summarizes the related parameters used in the model. The FSB has a bandwidth of 2.4 GB/s which is common for typical embedded systems. The PCI bus has a bandwidth of 133 MB/s assuming the width is 32 bits with 33 MHz clock speed. $L_{max}$ is defined as the maximum size of data that a bridge would allow before forwarding requests. Table 2 gives the list of logical data flows that we are going to consider. DM is a display manager which runs on the processor. We have used a large data size to produce significant delay even with a small number of flows (to keep the examples easy to follow). Parameter values of the flows are selected so that the allocation priority will follow the example that we have used in Figure 8. A posted write
PCI transaction does not always need a deadline other than when the age of a data matters. However, for delayed read transactions, it is important to make sure that the transaction is finished within its period. Delay is considered as the blocking time because the task has to wait for the data to arrive before doing the next job. Otherwise, it should not be a delayed read which may waste execution time. Considering the above, we have defined deadline values for flow 3 and 5.

### Table 1. Simulation parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSB bandwidth</td>
<td>( C_{fab} )</td>
</tr>
<tr>
<td></td>
<td>2.4 GB/s</td>
</tr>
<tr>
<td>PCI bus bandwidth</td>
<td>( C_{pci} )</td>
</tr>
<tr>
<td></td>
<td>133 MB/s</td>
</tr>
</tbody>
</table>

### Table 2. List of Logical Data Flows

<table>
<thead>
<tr>
<th>ID</th>
<th>Src</th>
<th>Dst</th>
<th>Size</th>
<th>Period</th>
<th>Deadline</th>
<th>PCI type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1</td>
<td>p3</td>
<td>1 MB</td>
<td>250 ms</td>
<td>N/A</td>
<td>PostedWrite</td>
</tr>
<tr>
<td>2</td>
<td>p1</td>
<td>p5</td>
<td>7 MB</td>
<td>250 ms</td>
<td>N/A</td>
<td>PostedWrite</td>
</tr>
<tr>
<td>3</td>
<td>p3</td>
<td>p5</td>
<td>3 MB</td>
<td>50 ms</td>
<td>100 ms</td>
<td>PostedWrite</td>
</tr>
<tr>
<td>4</td>
<td>p5</td>
<td>p2</td>
<td>1 MB</td>
<td>250 ms</td>
<td>N/A</td>
<td>PostedWrite</td>
</tr>
<tr>
<td>5</td>
<td>p3</td>
<td>DM</td>
<td>1 MB</td>
<td>200 ms</td>
<td>200 ms</td>
<td>DelayedRead</td>
</tr>
</tbody>
</table>

### 7.2. ASIIST User Interface

The layout of the ASIIST’s UI is presented in Figure 9. Once we start the tool, we can select the I/O configuration, if any, that we are interested in from the available list of modes on the left. Figure 9 is a screen shot taken for a model that uses I/O configurations. In our current example for the usage of the FFD algorithm, we do not use modes because we are changing the bus architecture. Loading the hardware flows for the specific mode and analyzing the delays are initiated by the buttons on the right side. ASIIST provides the amount of delay that a flow would experience while going through each bus segment. This also makes it easy to identify bottlenecks in the bus architecture so that a designer can make better decisions to correct infeasible designs. From now on, we will only show the table in the middle, which represents the hardware flow data and the result of the analysis.

### 7.3. ASIIST Usage with FFD Algorithm

Using ASIIST for the bus architecture in Figure 1 (a) gives the result shown in Figure 10. The deadline of flow 3 is missed, and the delay of flow 5 is larger than its period (i.e. its deadline). These deadline misses are caused by flow 3, where the data is sent from peripheral 4, located in the right side of the bus beneath PCI bridge 2, to peripheral 3, which is located in the left side. This long path gives a longer delay value for flow 3. Flow 5 is experiencing more delay than is needed because it is reading data from peripheral 3 which is sharing a bus with peripheral 5 and 1. The large bandwidth requirement of flow 2 is causing flow 5 to miss its deadline.

We will now briefly go over the FFD algorithm with the example model. The process is also depicted in Figure 8. For step 1, we identify the two flows that have the highest allocation priority. From Equation 2, flows 2 and 3 have the highest allocation priority. We create two sets of flows as shown in Figure 8 (b). The initiator and target are shown in the set. Step 2, we start adding the rest of the flows to one of the two sets. Flow 4 is added to set 1 because they have a common peripheral 5 (Figure 8 (c)). Flow 1 is

![Figure 9. ASIIST Layout](image)

![Figure 10. ASIIST Output for initial bus architecture](image)

![Figure 11. ASIIST Output for bus architecture made from FFD](image)
not possible to be assigned because its initiator and target are already in different sets. Flow 5 is not considered yet because it goes through the upper bridge. For step 3, we compare the two sets to decide which set gets direct access to the upper bridge. Since flow 5 is the only flow that goes through the bridge, the set that includes peripheral 3, set 2, is selected and the bus is extended as in Figure 8 (d). For step 4, we analyze the model with ASIIST to check if all the deadlines are met. The analysis result is shown in Figure 11 and because all the deadlines are met, there is no need to expand the bus architecture any more.

Following the steps of the FFD algorithm, we have derived the bus architecture shown in Figure 1 (b). From the result shown in Figure 11, the delay values of most of the flows, except for flow 1, have decreased considerably. Since flow 1 does not have a high allocation priority, it is the last flow to get an improvement out of using FFD for the given list of data flows. Flows 3 and 5 no longer miss their deadlines.

8. Related Work

In order to analyze a system model, we need an Architecture Description Language (ADL) which is able to specify the software logical architecture, hardware platform architecture, behaviors of the components, and assumptions of environmental components that are not directly specified.

Building an ADL to specify accurately all the properties of a system is very important because the analysis can only be as good as the model of the system. The Institute for Software Integrated Systems (ISIS) at Vanderbilt University has been developing a framework for this purpose, calling the approach Model Integrated Computing (MIC) [12]. MAST (Modeling and Analysis Suite for Real-Time Applications) [9] is another suite that defines a model capable of describing the timing behavior of a large set of real-time systems, including distributed systems and event-driven systems with complex synchronization schemes. These MAST descriptions are used as the input to the MAST tools for hard and soft real-time analysis.

An industry standard for modeling embedded systems has also been developed, called AADL (Architecture Analysis & Design Language), which is what we have used. AADL [8] is standardized by the Society of Automotive Engineers (SAE) and is defined in SAE Standard AS5506 [3]. Following this trend, more researchers as well as major industrial companies are starting to utilize this language for system development and analysis.

Open Source AADL Tool Environment (OSATE)\(^1\) is a set of plug-ins built on top of the open source Eclipse\(^2\) platform to give the front-end interface to build AADL models. Thus, tool developers can freely build and add new plug-ins that perform specialized analysis on AADL models. There are also several tool sets that support AADL, such as Ocarina [11], Cheddar [19], ANDES [17], ADAPT [18], etc.

Cheddar is a set of tools that performs scheduling simulations and feasibility tests for quick prototyping of real-time schedulers. Ocarina is a tool set which proposes AADL model manipulation, generates formal models, performs scheduling analysis and generates distributed applications. [20] presents a translation of AADL models into the real-time process algebra ACSR (Algebra of Communicating Shared Resources) that allows schedulability analysis of AADL models.

ADAPT is a tool which aims at facilitating the evaluation of various dependability measures (such as reliability and availability) from AADL models. It is based on model transformation rules, from AADL to Generalized Stochastic Petri Nets (GSPNs). ANDES is used for modeling a wireless sensor network system and analyzing its performance before deployment. It supports communication schedulability analysis, target tracking analysis and real-time capacity analysis. It extends AADL for the semantics needed for specifying sensor networks.

MARTE (Modeling and Analysis of Real Time and Embedded systems) [7] is a UML profile extension for real-time embedded systems which is standardized by the OMG (Object Management Group). It also provides an annex to relate to AADL based models. [13] further investigates how specific AADL concepts required for end-to-end flow latency analysis can be represented in MARTE.

In [22], the authors describe a tool that is able to derive and solve schedulability equations for a general communication and computation architecture. The tool is based on Real-Time Calculus [21], an extension of the network calculus theory [6] that we use in Section 4. However, the level of abstraction of the tool is much lower than in our approach: it requires the designer to model each communication element as one or more basic blocks, and then manually compose blocks to represent the whole system. This requires a deep understanding of the inner working of hardware elements and can be error prone for avionics systems comprised of hundreds of flows.

SymTA/S [10] is a tool that has been developed for system-level performance of real-time properties. However, they do not use explicit abstractions of bus protocols mainly because they are targeted for system-on-chip which uses heterogenous scheduling techniques.

Our work is different from others by the fact that, to the best of our knowledge, none of the tools incorporate the specific bus architecture design of the hardware platform from the architect’s perspective. Our tool, ASIIST, can ana-

---

1. Open Source AADL Tool Environment (OSATE) http://www.aadl.info
2. Eclipse http://www.eclipse.org
9. Conclusion and Future Work

This paper presents an automated tool, ASIIST, utilizing AADL to analyze bus communication performance for an arbitrary bus architecture and we apply this tool to an analysis of the PCI bus protocol. By no means will it replace rigorous testing, but it will enhance the designer’s capability to make more promising decisions. It is up to the designer to decide upon alternatives, but this tool will provide automation to quickly analyze a larger design space of bus architectures. The frequent changes in the schedulability equations caused by any simple alteration of the design is what makes ASIIST essential. Using this capability, we define an extensible framework for deriving application-specific design of I/O architecture using PCI as an example.

For our future work, we plan to develop more advanced algorithms that make better estimation of bus performance with stronger constraints. We will perform a benchmark testing with a real avionics system that has been implemented, to compare the estimated performance. With the recent new version release of AADL 2.0, some changes will be made to accommodate the new features in extending AADL.

Acknowledgment

We thank Peter Feiler, Jorgen Hansson and Dionisio de Niz of SEI for collaboration and assistance on AADL issues and John Mettenburg and John Glenski of Rockwell Collins Inc. for providing us with the sample problems and discussions of AADL models for bus architectures.

References